

API Management Patterns for Public, Partner, and Group Web API Initiatives with a Focus on Collaboration

Gloria Bondel
gloria.bondel@tum.de
Technical University of Munich
Garching, Bavaria, Germany

Andre Landgraf
Technical University of Munich
Garching, Bavaria, Germany

Florian Matthes
Technical University of Munich
Garching, Bavaria, Germany

ABSTRACT

API management of public, partner, and group Web APIs (APIs) is an organizational function at the interface between several stakeholders inside and outside of an organization. Most current API management literature is concerned with technical aspects of API management. Therefore, we use a design science approach to identify API management stakeholders, pattern candidates, and patterns focusing on collaboration. We derive these results from 16 expert interviews with API management team members, mainly working at established and SME organizations in Europe. The pattern's target audience are API provider team members. The purpose of this paper is to describe the design approach, the two exemplary patterns "Role-based marketing" and "Frontend venture", and six overarching observations made during the pattern language design. The first observation is that API provider teams usually control the resources used to collaborate with API consumers. Nevertheless, consumers want personal contact and contractual agreements before integrating an API. Also, most API management concerns are related to the API consumer. Then again, few standardized approaches for collaboration with backend functionality providers, legal or marketing exist. Additionally, an API has its own lifecycle. Finally, the strategic relevance of the API resources and the API consumers' organizational structure influence a patterns' suitability.

CCS CONCEPTS

• **Software and its engineering → Design patterns**; *Collaboration in software development*.

KEYWORDS

API Management, Collaboration, Boundary Resources

ACM Reference Format:
Gloria Bondel, Andre Landgraf, and Florian Matthes. 2021. API Management Patterns for Public, Partner, and Group Web API Initiatives with a Focus on Collaboration. In *European Conference on Pattern Languages of Programs (EuroPLoP'21), July 7–11, 2021, Graz, Austria.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3489449.3490012



EuroPLoP'21, July 7–11, 2021, Graz, Austria

ACM ISBN 978-1-4503-8997-6/21/07.
https://doi.org/10.1145/3489449.3490012

1 INTRODUCTION

Web Application Programming Interfaces[1] (APIs) are interfaces that expose their endpoints over the public internet and which developers can access using the HTTP protocol [5]. APIs enable the emergence of digital platforms as they allow third-party developers access to core platform functionalities [25]. In Information Systems (IS) research, APIs are conceptualized and analyzed as boundary resources, i.e., resources at the interface between platform owner and third-party developer [17, 19, 24, 25, 33]. Thus, APIs are resources with strategic value [52] and firms planning to establish or already operating a platform need to design them carefully.

Most research on APIs as boundary resources focuses on ex-post analyses' of successful platforms controlled by relatively young, digital organizations like Apple and Google [17]. Also, these young, digital organizations are more likely to provide public APIs, and those APIs are integrated more often by third-party developers than APIs of established organizations, i.e., organizations that were traditionally not digital [4, 21]. As an example, in [21], the authors compare the number of APIs and their use in mashups of the young, digital retailer Amazon and the established retailers Walmart and Macy's, showing that Amazon has more public web APIs and that more mashups use Amazons APIs. A study that aims at identifying barriers that hamper German automotive organizations from providing public APIs suggests that legal, economic, social, technological, or organizational barriers exist for established organizations [6, 7]. In the German automotive sector, especially unclear business models, strict regulation, and expensive data transmission are challenges [6]. Nevertheless, our experiences and interactions with established organizations and small or mid-sized enterprises (SMEs) in different sectors indicate that many of these organizations plan or already use public, partner, or group API initiatives. Such API initiatives aim to enable platformization [17, 19, 24, 25, 33], the realization of new business models [16, 37], efficient partner integration [32], or compliance (e.g., in banking).

The literature on APIs as boundary resources is concerned with APIs on a relatively abstract level that does not allow API provider teams to derive recommendations for action easily. We define API providers as all stakeholders carrying out API management tasks, i.e., activities for designing and maintaining APIs as well as additional technical and social resources to enable the API to be used by external consumers. More specific API management guidelines, including patterns, can be found in the practice-driven API management literature (see Sec. 7). Hence, most of the literature is concerned with rather technical aspects of API management, i.e., the design and maintenance processes of specific software components. To give just a few examples, endpoint design guidelines

[1] Henceforward, we use the terms *Web API* and *API* as synonyms.

are concerned with achieving RESTful compliance and security guidelines are concerned with implementing authentication and authorization mechanisms [16].

However, API management as an organizational function at the interface between the provider and public, partner, or group consumer makes knowledge transfer between different stakeholders necessary [31]. An API initiative cannot exist in isolation, but ongoing collaboration with various stakeholders inside and outside the organization is required. Nevertheless, to the best of the authors' knowledge, only few best practices for knowledge transfer and collaboration in API management have been explicitly formalized.

Overall, we aim at identifying API management patterns focusing on collaboration between an API management team and other internal and external stakeholders of a Web API initiative. The pattern language analyzes public, partner, and group API initiatives, meaning that the API provider and API consumer belong to different organizations. The resulting pattern language should be applicable for different types of API provider organizations, including established organizations and SMEs. To achieve this goal, we answer the following research questions:

- ***RQ1:*** *Who are the stakeholders involved in API management?*
- ***RQ2:*** *What are API management patterns concerned with collaboration between the API provider and other stakeholders?*

We create a pattern language by applying a design science research approach [28, 29]. The data collection consists of 16 interviews with API management team members describing tasks, issues, and solutions of their daily work. From these interviews, we generated a case base, i.e., a collection of cases each representing a distinct API initiative, holding 12 cases. The data analysis yields the documentation of eight stakeholders and 56 pattern candidates, of which 21 qualify as patterns.

These results contribute to practice since API providers can explore proven solutions for specific concerns, taking into account their respective API initiatives' characteristics. Additionally, they can use the pattern language to benchmark their practices with proven solutions. The pattern language furthermore provides a common taxonomy that can be used between API providers and API consumers as well as between IT and business stakeholders within an organization. The scientific contribution is the documentation of state-of-the-art collaborative activities between API management stakeholders that provides a basis for theorizing on collaboration and knowledge transfer within and outside of an organization.

In the following section, we introduce basic concepts of API initiatives to create a shared understanding as a basis for the pattern language. Afterward, we present the research approach in Sec. 3 and the purpose, target audience, and structure of the pattern language in Sec. 4. The definition of stakeholders is described in Sec. 5, followed by two exemplary patterns documented in Sec. 6. In Sec. 7 the results are related to the existing literature on API management. Sec. 8 discusses interesting findings made during the design of the pattern language. Finally, Sec. 9 summarizes the research endeavor and describes planned future work. Additional materials are provided in the appendix.

2 BASIC CONCEPTS OF API MANAGEMENT

In this section, we present the basic concepts and relationships between software artifacts in API management as illustrated in the lower part of Fig. 1. The aim of this model is to enable a clear and shared understanding as well as basic naming conventions for the pattern language, since a consistent taxonomy is an important aspect of a pattern language [9, 10, 34]. The model is derived from literature and interview data.

The API management software artifacts are a backend, an API gateway, a web API, an API developer portal, and the API-consuming application. The *backend* can be any software component that provides a particular capability, i.e., functionality or data, that should be made accessible to other stakeholders. The *Web API* is used to make the functionality or data of the backend accessible. A Web API is an interface available over the public internet and thus can be accessed using the HTTP protocol. An *application* integrates the functionality or data provided via the Web API. This application could itself also be a backend providing (enhanced) functionality or data to further applications.

Besides, two types of software platforms support API management, i.e., the API gateway and the API developer portal. The *API gateway* is a reverse proxy that intercepts incoming requests from clients. Hence, an API gateway is an infrastructure platform managing the API provider and consumer interaction at runtime. Management capabilities provided by the API gateway include authentication, rate limiting, statistics and analytics, monitoring, policies, alerts, and security [44]. The *API developer portal* is a web application that allows the API provider to communicate information to API consumers needed to find and use an API. This information should comprise the API documentation including information on the APIs location (URI) and the structure of valid API calls. Additional information can contain, among other things, terms and conditions, contact information, marketing information, changelogs, prizing information, and social content like forums or blogs [16]. Also, self-services supporting the API consumer can be part of the API portal, e.g., a self-service for API consumer registration [16]. Often, the API portal shares data with the API gateway, for example to ensure that only registered users can make requests to an API. Both API management software components, the API gateway and the API developer portal, are optional.

An *API initiative* denotes any new or ongoing endeavor of providing a Web API to potential consumers, irrespective of the use of an API gateway or API portal.

3 RESEARCH APPROACH

We apply a design science research approach according to [28, 29] as illustrated in Fig. 2 to answer the research questions. Design science is a methodology broadly applied in IS research for developing theory, models, and rich descriptions, especially in emerging research domains [51]. Design science research aims at creating purposeful artifacts [29]. The artifact that we create is a pattern language supporting API provider teams in API management activities focusing on collaboration with other stakeholders. A *pattern* describes an abstract solution to a problem that captures the core of the solution without prescribing any implementation details [2, 27]. Thus, a pattern can be reused and adapted to different contexts [10]. A *pattern language* addresses a complex problem that cannot be solved with a single pattern but instead presents a collection of related patterns [2, 38]. Each pattern of the pattern language defines a more specific problem and the respective solution within

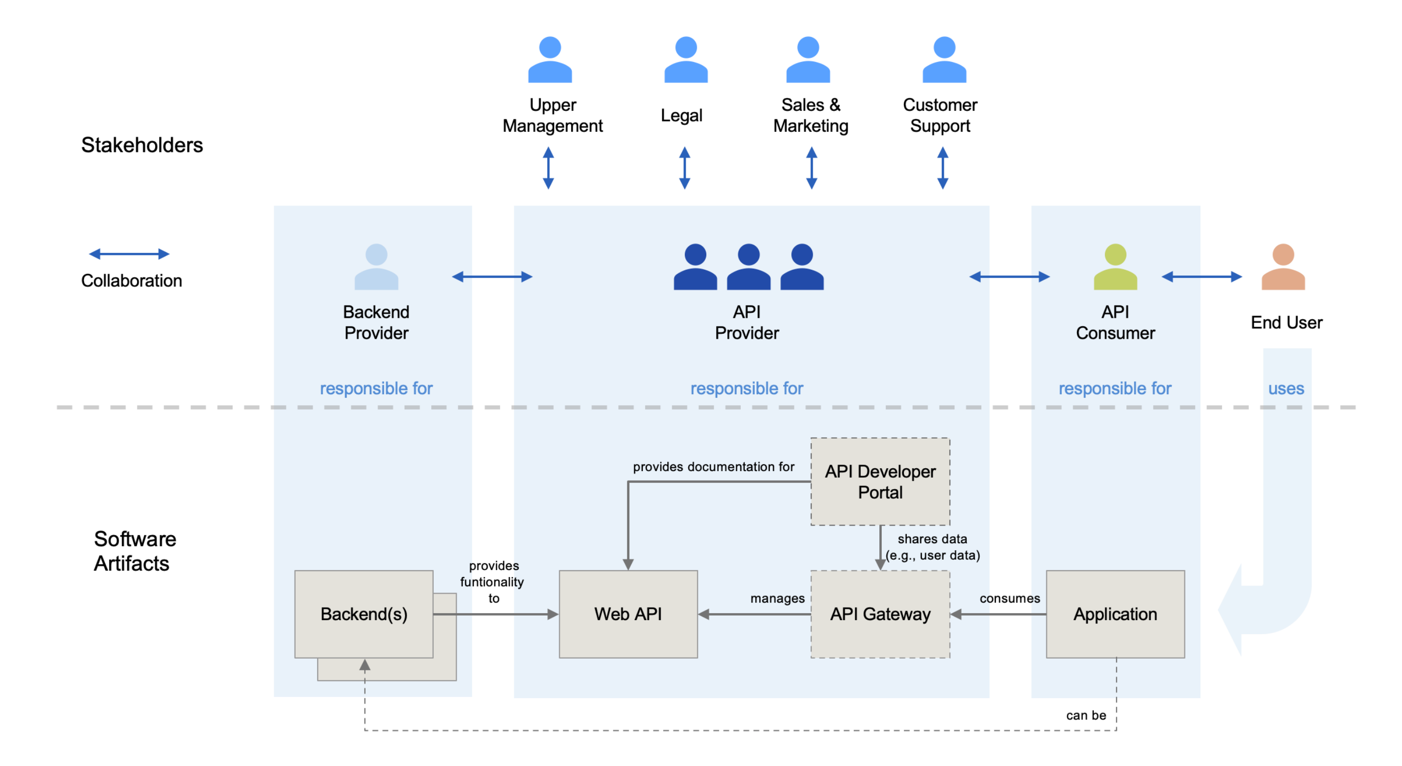


Figure 1: Conceptual overview of software artifacts in API management and stakeholders interacting with the API management team.

the context of the pattern language [38]. The purpose of a pattern language and each pattern is to enable knowledge dissemination in the respective domain [10]. Furthermore, pattern languages allow the user to solve a problem step by step instead of having to design an overall solution to a complex problem upfront [1]. More precisely, it provides practitioners with operational knowledge on current practices in a domain, and academia with knowledge on the evolution of a discipline [9, 10, 34]. In software engineering, pattern languages have already been successfully introduced in the past, e.g., by [12, 23].

3.1 Relevance Cycle

Relevance in design science ensures that a research endeavor contributes to solving a vital business problem in a particular application environment [28, 29]. Organizations use API initiatives to realize different goals, including platformization [17, 19, 24, 25, 33], the realization of new business models [16, 37], efficient partner integration [32], or compliance (e.g., in banking). We focus on API provider teams operating at the interface between functionality provider organization and consumer, aiming to attract and retain consumers of an API. Thus, API management should be treated as a highly collaborative task that requires a lot of communication, cooperation, and coordination [8] with functionality providers and API consumers. Furthermore, the API management function is embedded into an organizational context, which makes collaboration with additional stakeholders like the legal or the marketing department necessary. The importance of API management's collaborative aspects has also been confirmed in our interviews, mostly conducted with API provider team members of established organizations and SMEs in Europe. Thus, the pattern language's goal is to provide API management team members with support on designing the collaborative aspects of API management. The resulting pattern language will be made available for practitioners as a pattern catalog upon finalization.

3.2 Rigor Cycle

Rigor in design science is achieved by examining existing knowledge and applying it appropriately as well as applying sound methodologies [28, 29]. We examined IS literature on platforms focusing on boundary resources [17, 19, 24, 25, 31, 33, 52] and API management literature [16, 32, 35, 37, 41, 42, 47, 48, 53–55] (see Sec. 7) to inform the research endeavor. Furthermore, we inspected literature concerning pattern creation [1, 2, 12, 14, 23] and thoroughly applied design science [28, 29] and a grounded theory methodology [51] approach to design the pattern language as the resulting artifact. The pattern language informs researchers on the current state-of-the-art and changes in the API management domain. It further provides a basis for additional theory-building [9, 10, 34].

3.3 Design Cycle

An overview of the design approach for creating the API management pattern language is presented in Fig. 3. Since API management patterns should capture current best practices, we interviewed practitioners working in API management to collect data. We conducted 16 semi-structured interviews with 15 interviewees between August 2020 and January 2021. Out of the 16 interviews, 13 interviews

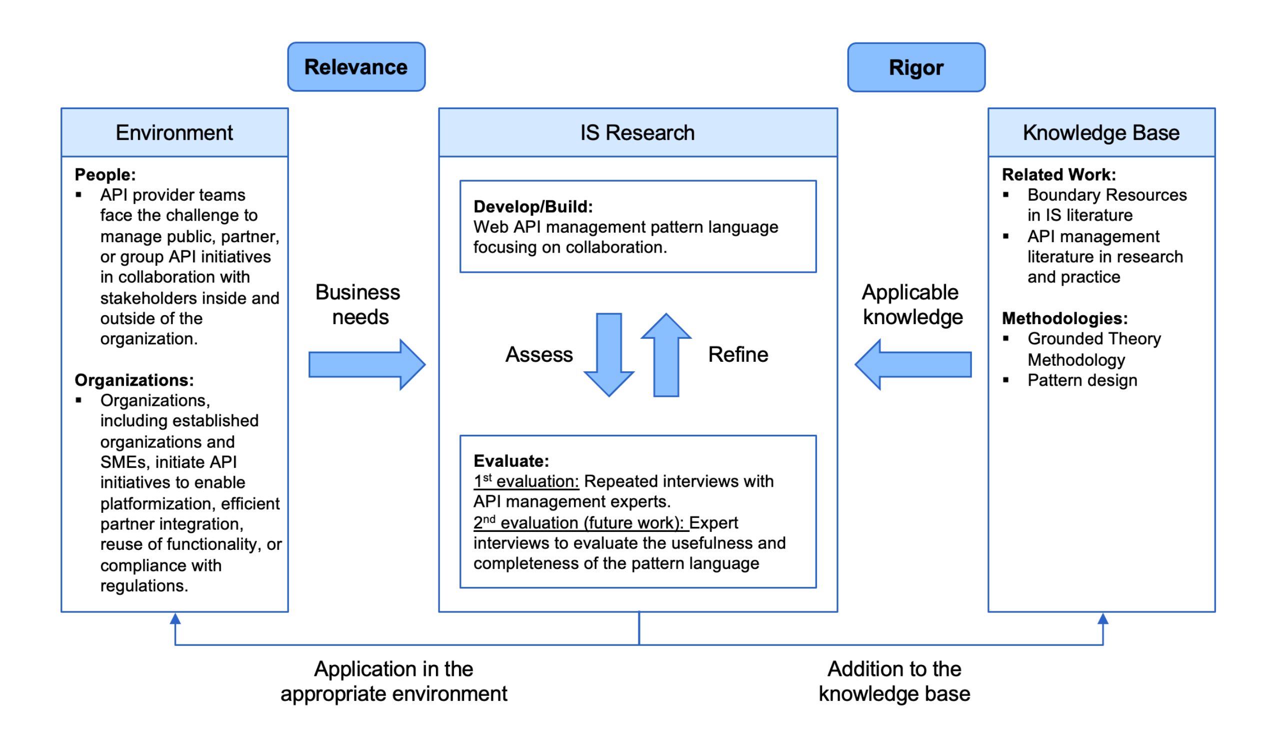


Figure 2: Information systems research framework [28, 29] adapted to creation of the API management pattern language focusing on collaboration between stakeholders.

were initial, and three were follow-up interviews with the aim to get more information on specific pattern candidates and evaluate first pattern ideas. The interviews focused on past and current tasks or issues that the interviewees face in their API management daily work. Furthermore, we discussed solutions to address the tasks or solve the issues successfully. We deliberately included as many diverse perspectives as possible, which is why we interviewed practitioners with different roles working in various industries at different sized organizations. An overview of the interviewees is provided in Appendix A.

In the following, we describe how we identified stakeholders, pattern candidates and patterns.

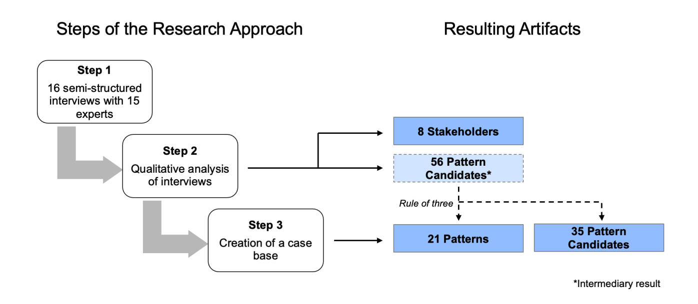


Figure 3: Overview of the research approach and the resulting artifacts.

Stakeholders and Pattern Candidates. In parallel to conducting the interviews, we analyzed the collected data by applying a Grounded Theory Methodology (GTM) approach, that guides qualitative content analysis in Information Systems (IS) research [51]. All interviews were transcribed and we defined two seed categories, which are *stakeholders* and *pattern candidates*. We used open coding to create new subcategories and assign codes to these subcategories. Additionally, we iteratively compared new codes to existing codes. If possible, we summarized these codes into new subcategories. This approach resulted in the identification of eight stakeholders and 56 pattern candidates.

Pattern. As a next step we analyzed which pattern candidates can be conceptualized as genuine patterns applying the *rule of three* [14] (also applied by [9, 34, 49]). The rule of three states that *"[...] a good pattern should have three examples that show three insightfully different implementations."* [14]. However, during the interview analysis, we realized that some interviewees talked about several API initiatives, making it difficult to map each interview to one pattern instantiation. We decided to identify and categorize all API initiatives described during the interviews as distinct cases to account for these observations. Hence, we collected the cases in a case base and assigned an ID containing the letter C and a number to each case to distinguish and reference them, e.g., case "C3". In the case base, we included each API initiative at the API portal level. The rationale for choosing this granularity level is that we observed that the APIs managed in one API portal all share the same general context regarding collaboration with the API

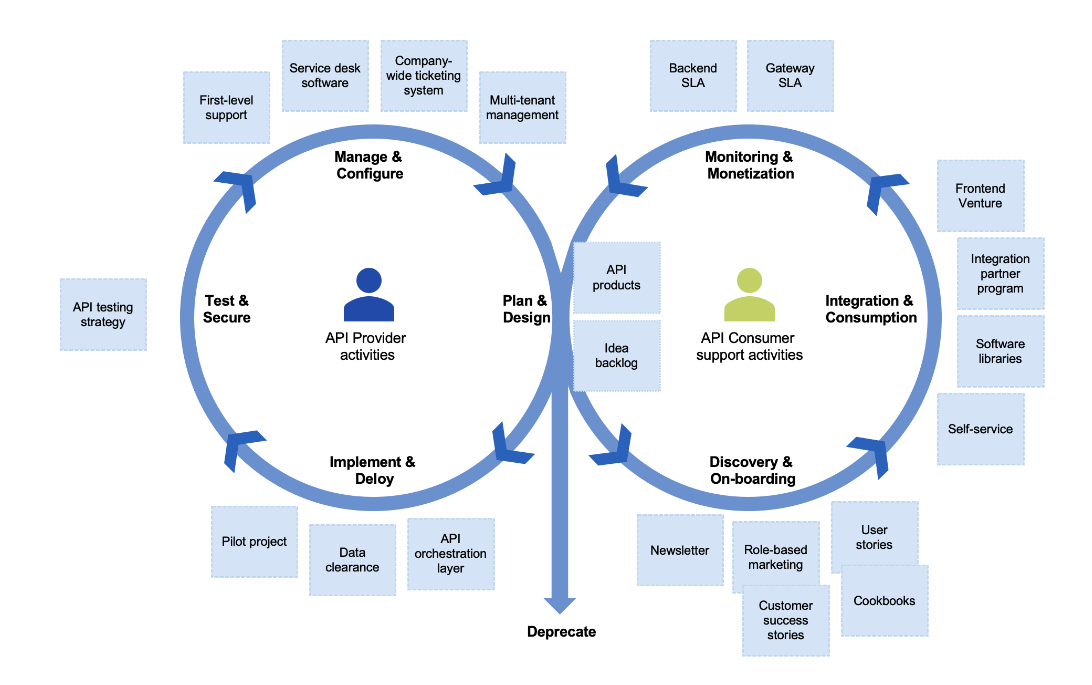


Figure 4: API management patterns structured according to an API management lifecycle.

consumers. For example, the provider of a public API portal with several endpoints and more than 10,000 external consumers could provide a contact form for API consumer inquiries. However, if the same organization also provides an API portal for few endpoints and less than five contractual partners, direct communication via email or phone could be more suitable. Thus, characteristics influencing the suitability of patterns like the type of users, the number of users, or the API initiative’s maturity level can differ for different API portals. Based on the interviews, we identified 15 unique cases. However, three of these cases describe internal API initiatives and are thus excluded from the analysis. As a consequence, we analyzed 12 cases. An overview of the case base is provided in Appendix B.

After creating the case base, we applied the rule of three [14] to the pattern candidates by counting how many cases in the case base actually implement a pattern candidate. We identified 21 patterns, and for each of these patterns, we thoroughly analyzed the instances and formulated all necessary sections.

Thus, the first version of the pattern language entails eight stakeholders, 35 pattern candidates, and 21 patterns. We fully document two of the 21 patterns in this paper (see section 6) and introduce patlets for the remaining patterns (see Tab. 1). All 21 patterns will be documented and published in a suitable outlet in the future. Together with the patterns, we will also publish all pattern candidates.

The next step in creating the pattern language will be to conduct further interviews with industry experts to validate the identified patterns and potentially promote additional pattern candidates to patterns. Furthermore, we plan on conducting expert interviews to evaluate the completeness and usefulness of the pattern language. Additionally, we are looking for industry partners who want to use the pattern language to benchmark their API management function or implement new patterns in case studies. A future goal is to publish results on the application of API management patterns.

4 PATTERN LANGUAGE SUMMARY

A pattern language summary describes the complex problem and its context that a pattern language as a whole tries to solve [38]. Furthermore, it introduces the patterns and their relations of which the pattern language consists [38].

The API management pattern language aims to support established organizations and SMEs to initiate and manage public, partner, or group API initiatives. The pattern language consists of management patterns focusing on knowledge transfer and collaboration instead of low-level, technical patterns. The primary target audience of the pattern language is stakeholders involved in the provision of APIs, i.e., API provider teams, in all organizations, but especially in established organizations and SMEs. The patterns are deliberately phrased to meet the information needs of business and IT stakeholders. Thus, API providers can not just apply specific patterns but also use the pattern language to bridge communication gaps between business and IT and create a common taxonomy. Additionally, other stakeholders can use the pattern language, e.g., API consumers searching for suitable collaboration patterns to propose to API providers.

We choose an API management lifecycle to provide the reader with an initial overview of the pattern language structure. There are many API management lifecycles put forward by practitioners in books and on the web, but there is no one API management lifecycle that research or practice agrees upon. Since we focus on API management as a function at the interface between two organizations, we use an API management lifecycle that explicitly describes activities of the API provider focused towards the API

consumer and with a focus on internal tasks [3, 36, 39]. Based on the API management lifecycles put forward by [3, 18, 26, 36, 39], we present the API management lifecycle illustrated in Fig. 4. The API management lifecycle iterates through all stages until the API provider decides to deprecate the API. The patterns are allocated to the respective phases that they support. The following phases are part of the API management lifecycle:

- *Plan and Design:* Decide to create or evolve an API and define the business goal, target market, and use cases. Create the API specification entailing the API endpoint structure and behavior. The phase can also include mocking to simulate the behavior of the designed API.
- *Implement and Deploy:* Realize the API using agile methods, e.g., by aggregating and abstracting existing backend functionality. Integration frameworks can support the implementation. Also, transition into production using Continuous Integration/Continuous Deployment (CI/CD) pipelines.
- *Test and Secure:* Test an API's behavior to validate the implementation of functional and non-functional requirements. Enforce policies and other security measures to ensure that adversaries cannot exploit the API.
- *Manage and Configure:* Resolve defects and issues, implement change requests and manage versioning. Configure the API management platform according to the use case, e.g., with regards to access rights and authentication methods.
- *Discovery and On-boarding:* Allow API consumers to discover and understand the value of your API using marketing initiatives and documentation. Provide on-boarding and testing facilities, e.g., self-services or sandboxes.
- *Integration and Consumption:* Support API consumers with the integration of APIs into their system landscape with the help of additional resources, e.g., documentation or SDKs. Ensure that your API runs reliably so that API consumers are satisfied with consuming the API. Provide support channels for API consumers in case of defects or issues.
- *Monitoring and Monetization:* Analyze the performance and usage of the APIs and monetize accordingly.
- *Deprecate:* Announced the APIs deprecation to current consumers and no longer evolve or supports the API. At some point, switch the API off.

Furthermore, the reader can gain a short overview of each pattern as a patlet in Tab. 1. A patlet presents a brief description of an API management pattern's content. The detailed description of all patterns is currently being revised and will be published in a suitable outlet in the future.

Finally, the relations between all patterns are illustrated in Appendix C. The visualization originates from the relations described in the *"Related Patterns within the Pattern Language"* section of each pattern. Also, the figure aids API providers and consumers find a logical "entry point" into the pattern language. An API provider planning an API initiative can start by reviewing or applying the pattern *"Idea Backlog"* and move on to related patterns. The API consumers most likely get in contact with an API via the API documentation, which is why the API consumer should explore the API management pattern language starting with the pattern *"User Stories"*.

5 STAKEHOLDERS

We identified eight stakeholder groups as illustrated in Fig. 1. We visualize the collaboration between the API provider and other stakeholders. While the other stakeholders also communicate, we did not illustrate those collaboration flows for conciseness reasons. Also, a stakeholder role does not have to be occupied by one or more dedicated persons. Instead, a person could have several roles.

The *API provider* is responsible for carrying out API management tasks. The API management comprises all tasks related to designing and maintaining the Web API, the API gateway, and the API developer portal. An API provider team includes business and technical roles, with business roles defining and pursuing business goals and technical roles aiming to ensure technical KPIs [37]. The tasks of an API provider comprises all activities aimed towards realizing the goals defined by the API management lifecycle. In many cases, backend developers also design and maintain the respective Web API and thus occupy two roles.

The API provider team also collaborates with the *API consumer*. The API consumer is the team that integrates the Web API into its application. Such an application can be either an end-user application or an application that other API consumers use, e.g., an API wrapper. Thus, communication can include inter alia passing on additional information on the API functionality, change requests, or issue reporting. The API consumer can belong to a different organization than the API provider team. The differences in the organizational affiliation between the API provider and the API consumer are discussed in the literature [16, 32, 46]. Accordingly, if the API provider and the API consumer belong to the same organization, the API is categorized as a private API. If the actors belong to different organizations, the API can be a partner or a public API. Partner APIs are accessible only for selected external partners, while public APIs are accessible to every interested developer. The presented API management pattern language focuses on partner and public APIs. The *end-user* denotes the person using the application and thus mainly communicates with the API consumer who provides the application.

The API provider also collaborates with the *backend provider* who designs and maintains the backend. Here, the collaboration mainly focuses on change requests and bug reporting for backend functionality that the API consumer or monitoring tools report to the API provider team. The API provider and the backend provider often belong to the same organization, thus the organization makes its own functionality or data accessible to external consumers. However, there are two settings in which the API provider and the backend provider belong to different organizations. In the first setting, the API provider operates a marketplace, that allows other backend and API providers to publish APIs. Still, all API marketplaces that are part of our case base also offer APIs provided by the API providers organization. Secondly, we observed two cases in corporate group settings where the API provider is employed by one subsidiary firm focused on IT provision, and the backend providers are other subsidiaries in the group. Nevertheless, the groups' overall goals still guide the interaction between the backend provider and the API provider.

Additionally, several other stakeholders can interact with the API provider team. First of all, the *upper management* can support

Table 1: Patlets of the API management pattern language focusing on collaboration.

Pattern Name	Short Pattern Solution Description
Company-wide ticketing system	A uniform ticket system allows the API provider organization to handle the defect communication in a unified manner across different teams of an organization. In case of an issue, the respective stakeholder creates a ticket that the system forwards to the team with ownership of the defect component.
API testing strategy	A central unit defines a testing strategy that ensures the coordination of the testing efforts of all API management teams. The teams use testing to reduce the likelihood of unexpected behavior of new or changed APIs or backends.
Pilot project	The API provider designs the API iteratively in close collaboration with the API consumer in a pilot project. Hence, the API consumer communicates needs and requirements and provides feedback to the API provider before, during, and after the API implementation project.
Frontend venture	The API provider enable API consumers that cannot, for any possible reason, integrate an API, to still use the API with a simple stand-alone frontend. If enough consumers are interested, a product team takes over the development and maintenance of the product.
Backend SLAs	An SLA is an agreement between two parties that specifies the quality of services provided by a party as well as contractual punishments in case of SLA breaches. In the case of backend SLAs, the backend and API providers agree to provide APIs with a specified level of quality to the API consumers.
Gateway SLAs	An SLA is an agreement between two parties that specifies the quality of services provided by a party as well as contractual punishments in case of SLA breaches. In the case of gateway SLAs, the API gateway provider agrees to provide API gateway services with a specified level of quality to the API consumers and provider teams.
Data clearance	A data clearing process ensures that all API endpoints comply with legal and strategic requirements before it is published to third-party API consumers.
API orchestration layer	An API orchestration layer abstracts the invocation of several backend services into a single API. The API orchestration layer thereby supports the tailoring of APIs that fit the user stories of the API consumers.
API products	The API provider bundles APIs that together address a business need and defines a monetization scheme to create an API product. API products can have different non-functional properties or include additional services.
Idea backlog	An idea backlog is a dynamic list that stores and organizes consumer wishes derived from consumer support requests, discussions, or surveys. The API provider uses the idea backlog to aggregate and abstract the consumer wishes to design new APIs or evolve existing APIs that meet consumer needs.
User Stories	The API provider documents API products and uses cases from a consumer perspective as user stories addressing a consumer's business need.
Cookbooks	A cookbook is a recipe-like, step-by-step integration guide for one user story. The cookbook describes the API integration from a consumer perspective. The overall goal is to guide the user from start to end without the need to follow additional documentation.
Software libraries	API management provides consumers with code utilities to support the API integration. These code utilities can be libraries, i.e., wrappers around the APIs and allow the API consumer to invoke them [16]. Thus, the application consumer does not interact with the API calls directly but indirectly through the library functions.
Integration Partner Program	API providers create and maintain a curated list of integration partners that meet specific quality criteria on the API portal. Integration partners offer the implementation and other services related to the API.
Role-based marketing	Role-based marketing denotes the design, maintenance, and clear separation of marketing material and other consumer-facing resources in the developer portal targeted at different user roles.
Newsletter	The API provider summarizes changes to existing APIs and new APIs in a newsletter. Newsletter management comprises the management of contact information, newsletter creation, and newsletter distribution.
Customer success stories	An API provider can demonstrate an APIs potential using success stories. A success story documents a past, successfully finalized product implementation project using the APIs.
First-level support	First-level support is the first point of contact for external software consumers. Allowing API consumer to also use the first-level support reduces the number of inquires to the API provider team.
Service desk software	Service desk software provides capabilities to create and manage support tickets. A ticket is used internally to manage the request and to communicate progress and support to the API consumer. Each support ticket captures all relevant information for the individual support case.
Self-service	Self-service automates (parts of) the API registration and onboarding process. Full self-service allows the API consumers to register themselves and their applications to the API platform without the need to interact with the API management team.
Multi-tenant management	Each tenant is granted an own instance of the portal on which the tenant can publish APIs and configure its portal API management in isolation. The infrastructure of an internal API platform can be maintained by one central gateway/infrastructure team.

the API provider, e.g., by promoting API management in strategic initiatives. The API management team needs to involve the *legal department* to ensure privacy conformance of APIs. Moreover, the legal department can negotiate contracts with partners using the APIs. *Sales & Marketing* supports the marketing activities for new APIs or enforces compliance with corporate identity specifications. Finally, an existing *customer support* can be a contact point for API consumers, which then forwards tickets to the API provider team. All these stakeholders belong to the same organizations as the API provider.

Our pattern language focuses on collaboration patterns for the API provider.

6 EXEMPLARY PATTERNS

In this section, we present two exemplary patterns of the pattern language. These pattern are *Frontend venture* and *Role-based marketing*. An overview of all patterns is provided in the patlets overview in Tab. 1. We are currently still writing these patlets up, and we plan to publish the finalized patterns.

The structure of the pattern description follows pattern structure best practices as presented by Meszaros and Doble (1997) [38] and Buschmann, et al. (1996) [12]. In the sections *"Other Related Patterns and Related Literature"* of each pattern description, we reviewed the following API pattern publications and API management books to identify related patterns and approaches: [16, 32, 35, 37, 41, 47, 48, 55]. Within each pattern, we refer to related cases from our case base with their IDs as listed in Tab. 3 (e.g., the third case in the case base is referred to as "C3"). In some patterns, we also refer to publicly available information on API initiatives within or outside of the case base.

6.1 Pattern: Frontend venture

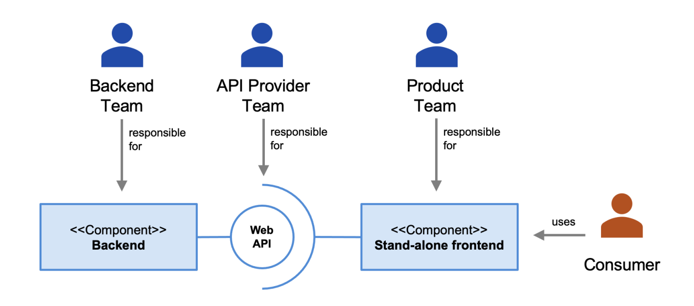


Figure 5: Visualization of the pattern *Frontend Venture*.

Stakeholders:

The *API Provider* has to collaborate with the *Backend Provider* and the *API Consumers* to realize this pattern.

Context:

Integrating an API into an existing IT landscape creates effort for the API consumers. However, some API consumers lack technical capabilities or the budget for API integrations and can thus not realize beneficial use cases. Such consumers are often municipalities or small, non-digital businesses.

Concern:

How can an API provider enable API consumers that cannot, for any possible reason, integrate an API to use the API still?

Forces:

- *Consumer capabilities.* Some potential consumers, especially municipalities or small, non-digital organizations, are not able to use or integrate APIs since they have no or small IT departments that are already working at capacity. For example, in one observed case, the API consumers are small, non-digital organizations that usually only have a website based on a content management system maintained by freelancers. Furthermore, these potential consumers often have no budget to hire external IT service providers to execute the integration project.
- *Legal obligation.* An API provider can be legally obliged to make specific data available to certain consumer groups through APIs, even if some consumers cannot use the API. Such legal obligations exist, for example, in the banking or automotive industry.
- *Effort.* The API provider has to design, implement, and maintain the stand-alone frontend.
- *Quality.* The API consumers use a stand-alone frontend only if the API provider can provide it with a certain level of quality.
- *Public perception.* The API provider can make data and functionality available to municipalities or small, non-digital businesses in a way that supports some societal interest. Thus, the public might view the API provider's efforts positively.
- *Profits.* In some cases, the API consumers might be willing to pay for the use of the stand-alone frontend.
- *Reusability.* The API provider has to balance the specific needs of the first API consumer with the reusability of the frontend for other consumers.
- *Limited Functionality.* The frontend can implement all or just a subset of the functionality of the API.
- *Enhanced Functionality.* The frontend can enhance the responses of the API, for example by creating visualizations.
- *Limited flexibility.* An API and an interface provide different flexibility with regards to integration, automatizing, customization, and branding.

Solution:

The API provider identifies and evaluates a use case with an API consumer or a consumer group and implements a simple stand-alone frontend, i.e., websites with fields and buttons that trigger API functionality. The API provider develops the frontend in collaboration with the consumers. As soon as the frontend reaches a certain level of maturity, the API provider markets it to other potential consumers. If enough consumers are interested, a product team takes over the development and maintenance of the product. Thus, implementing a frontend can be a venture opportunity.

Implementation Details:

Approach. A stand-alone frontend implements a specific use case for a consumer or consumer group. Therefore, the first step is to analyze the need of the future consumer or consumer group. A potential tool that supports the use case definition is the Value

Proposition Canvas[2] [40]. Also, the API provider can apply the pattern *Tailoring APIs to products* and bundle use cases into products.

Once the use case is defined, the API provider assesses the benefits and drawbacks. The benefits can include additional direct profits from billing the consumers, the chance of further profits from reselling the frontend to other external consumers, internal reuse of the frontend, or positive marketing impact. The API provider has to weigh these advantages against the additional effort required to develop and maintain the frontend. Furthermore, as part of the initial analysis, the API provider should check if similar frontends already exist within the organization. If an internal team already built a similar frontend, the API provider can reuse (parts of) it. Based on this benefit-cost evaluation, the API provider decides if or how to implement the frontend.

Assuming the API provider decides to move forward with the frontend implementation, in the next step, the API provider develops a prototypical frontend either from scratch or based on an existing internal frontend. Iterative development and close collaboration with the API consumer as described in *Pilot Project* ensures that the frontend meets the consumers' needs. An alternative approach would be to implement a simple first version of the frontend in the course of a *hackathon*[3]. In both cases, the API provider has to find a balance between meeting the specific needs of the initial API consumers and the reusability of the frontend for potential future consumers.

After the frontend reaches a certain level of maturity, the API provider can present it to other interested parties. The API consumers need to have means to provide feedback and report issues, which can be realized with the pattern *First-level support*. Also, the APIs product page and the frontends product page should reference each other to increase each other's discoverability. If enough consumers are interested in using the frontend, the API provider can hand it over to a dedicated product team to evolve and maintain it as a standalone product. The API provider can inform other interested parties about the new frontend using the pattern *Newsletter*. Additionally, if the API provider wants to allow many API consumers to easily register for and access the frontend, they can implement the pattern *Self-service*. Thus, designing and implementing a frontend is a venture opportunity for the providing organization. The development team then acts as an IT provider to the consumer while the API platform provides the underlying API services.

Types of frontends. A stand-alone frontend realizes a specific use case and typically concentrates on a subset of the APIs functionality. In general, the goal is to provide a user interface using state-of-the-art design elements that consumers without technical capabilities can easily and intuitively use. The most basic type of a frontend is a user interface that simply makes the as-is API functionality usable for non-technical consumers. For example, the API provider can implement a website that allows users to upload data, set transformation parameters and response filtering options, and subsequently download the file containing the APIs response. A more advanced frontend can also implement some logic that augments the response data with additional data or visualizations. As an example, a frontend can show a map and locate certain events on it.

[2]The value proposition canvas is a tool to systematically define a value proposition, i.e., the benefits a solution creates for a customer or customer group. The value proposition canvas relates customer jobs, related pains, and potential gains with the provider's solution. It forces the provider to make explicit how the solution relives pains and creates gains for the consumer. Thus, it tries to ensure that a provided solution actually meets a need of the target consumer.

[3]A hackathon is an event with a pre-defined timeframe during which small development teams compete to implement the best solution to a pre-defined problem.

Consequences:

Benefits:

- *Consumer capabilities.* The API provider can make data and functionality available to organizations with insufficient IT capabilities or budgets. Still, other interested API consumers with enough IT capabilities can consume the API to create custom integrations or frontends.
- *Legal obligation.* In case of a legal obligation, the API provider has to implement the API anyway. Therefore, the API provider might view the legal obligation as an opportunity to create new business relationships or profits through the stand-alone frontend.
- *Public perception.* It results in good publicity for the API provider to make data and functionality available to municipalities or small, non-digital businesses in a way that supports some societal interest.
- *Profit.* The API provider can monetize the implemented stand-alone frontend directly, and it can thus become a source of profit. If the API provider does not provide the frontend, a third party could skim these profits.
- *Limited Functionality.* The API provider can decide to limit the functionality that the frontend provides compared to the API to guide the consumer with difficult tasks. Also the API provider can neglect rarely used endpoints.
- *Enhanced Functionality.* The API provider can enhance the user experience for the consumers if they enhance the responses of the API, e.g., using visualizations.

Drawbacks:

- *Effort.* The design, implementation, and especially the maintenance are additional efforts for the API provider. The API provider has to ensure that enough resources are available to realize a frontend venture.
- *Quality.* The API provider has to identify and maintain the level of quality that API consumers expect from the stand-alone frontend. If the stand-alone frontend is of insufficient quality, the API consumers will not use it, and it would thus be a loss of investment.
- *Reusability.* The API provider has to put effort into balancing the needs of different stand-alone frontend consumers. If the API provider tailors the stand-alone frontend too much to the needs of one API consumer, other API consumers will not use the frontend. However, if the frontend is too generic, it might be of less value for all consumers.
- *Limited Flexibility.* A frontend limits the flexibility compared to an API with regards to integration, automatizing, configuration, and branding. Thus, it is important that the API provider also keeps providing the API.

Related Patterns within the Pattern Language:

Within this pattern language, five patterns are related to the pattern *Frontend venture*. First of all, the API provider can use the pattern *API product* to define the use case that the frontend venture

implements. Furthermore, the pattern *Pilot project* can be used to develop a prototypical frontend. After the frontend has reached a certain level of maturity, API consumers can use the *Self-service* to access the frontend. In case of issues, the consumers can report issues with the frontend via the *First-level support*. Finally, the API provider can use the pattern *Newsletter* to communicate the development or changes of the frontend to interested parties.

Also, an API portal provider can use *Frontend venture* in combination with the pattern candidate *Hackathons*. A hackathon provides a good opportunity to develop an initial prototype of the stand-alone frontend.

Other Related Patterns and Related Literature:

The *Frontend venture* needs to ensure that only authenticated users can access the frontend. [32] and [16] describe authentication methods including *usernames and passwords* and *OAuth*.

A tool that is not described within this pattern language but can support the use case definition is the Value Proposition Canvas [40].

Known Uses:

We observed the pattern in three cases:

The API portal provider of an automotive organization (C3) wanted to provide data to two municipalities via APIs. However, the municipalities did not have the capabilities to integrate the APIs and requested a stand-alone user interface that a non-technical stakeholder can use. The API provider implemented such an interface during a pilot project since the organization expected positive marketing effects. After the pilot phase, the API provider team handed the prototype over to a product team that now maintains the frontend. As a result, the API provider reports that only 50% of consumers directly access the API behind the stand-alone frontend, while 50% use the frontend.

The organization C7 is an insurance subsidiary that provides insurance services within a group setting. The API management team offers a mix of APIs and frontends. If the API provider exposes an API directly or uses a frontend depends on the nature of the product. The products that need much integration into backend systems, customization, or branding are exposed as APIs. Additionally, the API provider provides frontends for products that do not need much integration, primarily for simple functionality or data access.

Finally, in C9, the organization offers a marketplace for IoT applications. The API provider again offers APIs and frontends, however, frontends are dominant. According to the interview partner, most consumers prefer frontends. However, the API provider also provides APIs to enable integration and automatizing.

Cross-case observations:

All the cases are in early phases (pilot phase or early production phase). This makes sense, since it can be beneficial for API initiatives in early stages to implement frontends to attract first consumers. However, it is also essential to maintain the APIs since APIs provide more flexibility regarding backend integration or frontend customization to organizations with more IT capabilities or budget.

Also, not surprisingly, the consumers are rather small business or municipalities.

6.2 Pattern: Role-based marketing

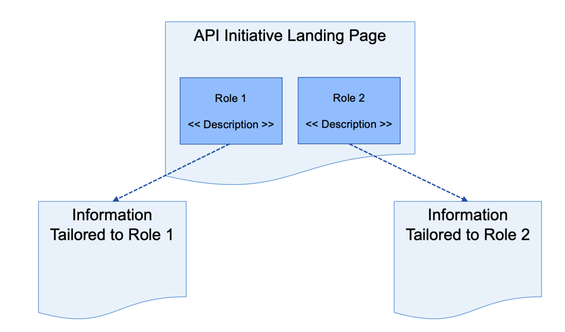


Figure 6: Visualization of the pattern *Role-based Marketing*.

Stakeholder:

The *API Provider* has to potentially collaborate with *Sales & Marketing* to realize this pattern.

Context:

API portals can target different types of user roles. Especially in established organizations, technical and non-technical stakeholders are involved in buying an API. Similarly, in marketplace settings, the API provider can differentiate between platform users and third-party developers.

Concern:

How can an API portal provider address the information needs of different API stakeholder groups?

Forces:

- *Decision-making*. The API provider offers an API that is of strategic relevance for the API consumer. Integrating a strategically relevant API has far-reaching consequences since the consumers' business relies on the API. Thus, business stakeholders like business owners or upper management want to be involved in the buy decision. The procurement of a consumer organization can be another non-technical stakeholder involved in a buying decision. However, the business stakeholders need to understand an API before buying but are often overwhelmed by technical specifications.
- *Technical information*. The API provider has to convince developers of the technical capabilities of an API. Developers need technical specifications to understand and work with an API, but they do not want the API provider to flood them with business-related marketing information.
- *Consumer types*. In marketplace settings, not only the user of the platform has to understand the product, but also third-party developers that want to create extending modules for the platform. These two roles have different goals and often also different IT capabilities. While third-party developers are usually tech-savvy, the main business of platform users is not always technical. Therefore, they also have different information needs.
- *Effort*. The realization of this pattern entails the creation and maintenance of two different documentation types.

Solution:

Role-based marketing denotes the design, maintenance, and clear separation of marketing material and other consumer-facing resources in the developer portal targeted at different user roles. Usually, these roles are a developer and a business role or platform users and third-party developers in a marketplace setting. The API provider tailors the resources to the information needs of the respective roles, potentially in collaboration with the sales & marketing department. The different types of resources have to be clearly distinguished and easily navigable to ensure that additional information does not interfere the other role's user journey.

Implementation Details:

Landing page. The landing page describes the purpose of the overall API portal and provides links to the product pages. In *Role-based marketing*, the landing page allows a visitor to choose between different views, e.g., a technical and a business view, or a platform consumer and a third-party developer view. Each role should carry a short description of the associated role and information.

Views. Each view contains information relevant to the stakeholder in a language that the respective stakeholder is accustomed to. The technical view focuses on the needs of developers, for example, by describing the user journey from registration to deployment, providing links to documentation early on, and reducing marketing material to a minimum. Furthermore, the resources that the API portal provider compiles for developers should include technical specifications, code samples, and technical illustrations. The information can also include the pattern *User Stories*.

On the other hand, the business view addresses business users and presents more abstract information on potential use cases, lists advantages of using the platform, provides pricing information, and includes marketing material, e.g., the pattern *Customer Success Stories*. Additionally, the view could include the pattern *Integration Partner Program* to point business stakeholders towards integration partners.

Similarly, in a marketplace setting, the view of a third-party developer focuses on technical documentation that allows the organizations to develop modules, sell them, and analyze their success.

Finally, the platform user view is mainly concerned with describing the platform's business value and additional services for the platform user. Both views can include the pattern *Customer Success Stories*. The API provider can present the third-party developers with examples of successful modules developed by other third-party developers, and the platform users can view successful cases of platform usage.

Content creation. The API provider can use the concept of *Personas*[4] [13, 43] to get a clear picture of the information requirements of different roles. Furthermore, the API provider should collaborate with several internal stakeholders to ensure that the content created for the different roles meets their respective expectations. First, the API provider should create the technical documentation. Secondly, the API portal provider should activate the business units responsible for the API to develop the resources for the respective stakeholders. Furthermore, the API provider should involve Sales & Marketing in the design of marketing materials. All resulting resources have to be cross-checked to ensure consistency.

Consequences:

Benefits:

- *Decision-making.* Information tailored to business stakeholders allows the business stakeholders to make better decisions. This is especially important in settings where adapting an API is of strategic relevance since it is difficult to replace the API later.
- *Technical information.* Separating the documentation into business and technical information makes it easier for developers to identify relevant information. The developers do not need to read any marketing information and can directly dive into technical documentation.
- *Consumer types.* Role-based marketing allows the API provider the address the information needs of different roles in marketplace settings.

Drawbacks:

- *Decision-making.* The API consumers business stakeholder will potentially use the dedicated online documentation only for discovery. The actual buy decision usually happens only after personal contact and contract negotiations since the API consumer wants to create close ties and get service guarantees before choosing to use a strategically relevant API.
- *Effort.* It is laborious to create and maintain developer information and marketing resources, especially if they reference each other. The API provider must ensure that the organization has enough resources to keep the documentation and marketing material up-to-date, especially if they release new API versions. Outdated documentation can lead to negative experiences for all stakeholders.

Related Patterns within the Pattern Language:

Within this pattern language, the API provider can use the pattern *Customer success stories* to create marketing content for business stakeholders of an interested API consumer, platform users or third-party developers. Similarly, the pattern *Integration partner program* can entail information relevant for business stakeholders and platform users. On the other hand, the pattern *User Stories* focuses on the information needs of developers and third-party developers.

Also, an API portal provider can use *Role-based marketing* in combination with the pattern candidate *Smart contact form*[5]. If the portal provider uses separate landing pages according to *Role-based marketing*, the contact form on each of the pages can be pre-categorized into business and technical or platform user and third-party developer requests.

Other Related Patterns and Related Literature:

The pattern *API Description* presented in [35] addresses the question of what knowledge to share between API provider and API consumer and documents this knowledge. While the described solution already comprises technical and business-related information,

[4]The concept of personas has emerged in the field of user-centered software design and denotes *[...] descriptions of imaginary people constructed out of well-understood, highly specified data about real people"* [43]. The advantages of using Personas are that assumptions about users are made explicit, it allows the designer to focus on a specific type of users, and they create empathy and a shared understanding for the user [43]. Thus, the designer can address the needs of a user more targeted.

[5]A smart contact form enables an API consumer to categorize a request into different categories, e.g., business and technical, and routes the ticket to the right contact within the provider organization.

the pattern *Role-based marketing* specializes the pattern *API Description* by defining technical and business contents as well as different marketplace roles as relevant and describing a way to structure the information.

According to [32], an API provider should use a developer segmentation analysis followed by force-ranking priority segments to identify the target audience consisting of product or technical teams. While technical and business stakeholders are not separated, the API portal provider can use these approaches before applying *Role-based marketing* to narrow down the general target audience, e.g., industry or type of organization.

Known Uses:

We observed the pattern in three cases:

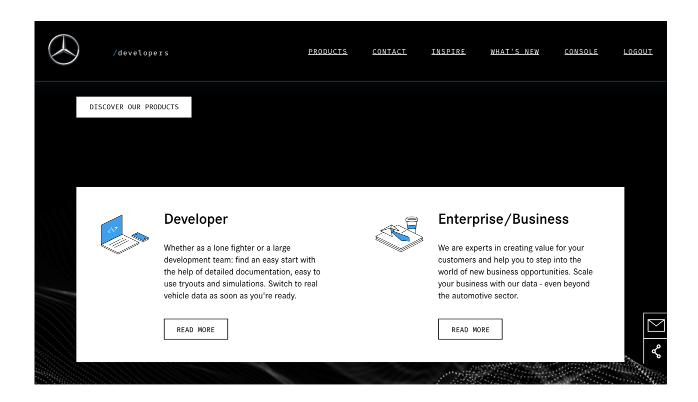


Figure 7: Role-based Marketing in the API Portal of Mercedes-Benz. (https://developer.mercedes-benz.com/ accessed 10/08/2021)

First, Mercedes-Benz offers API products and services to partner and third-party developers on its developer portal[6]. Fig. 7 shows the landing page. The users are immediately asked to select a role, either developer or enterprise/business. Thereby, the API provider assures that both user types are only one click away from customized marketing material.

In C9, the organization offers a public marketplace for IoT applications. The organization provides a core platform software, and third-party developers can build additional modules for the platform using APIs. Users of the platform can buy the core platform software and add pre-integrated modules according to their needs. The API marketplace offers dedicated documentation for two roles; users of the platform and the third-party developers that create additional modules for the platform.

Finally, in C13, the setting is the same as in C9, except that the platform is specific to the insurance industry. Again, the marketing material is aimed at two groups which are the users of the platform software and the third-party developers.

Cross-case observations:

All these cases have in common that the API initiatives offer APIs to partners and the public. Also, the initiatives are in production and have 1-100 consumers. The dominant monetization strategy uses individual contractual agreements.

These characteristics are in unison with the observation that the APIs offer industry-specific functionality with a strategic impact on a consumer‘s business as opposed to commodity functionality. Strategic functionality is difficult to replace once the API consumer integrates it into its landscape and business plan. Thus, it makes sense that business stakeholders of the consumer organization are steering or at least involved in the decision of buying. Also, the API consumer relies on the API provider to deliver functionality with business impact, which makes close ties with the provider organization and service guarantees essential. Hence the parties create individual contractual agreements. The provision of strategic functionality for specific industries further limits the number of potential consumers, explaining why the API initiatives have between 1-100 consumers.

7 RELATED WORK

This section distinguishes the API management pattern language with a focus on collaboration from related pattern languages for API management or software engineering.

Closely related is the microservice API pattern languages put forward by [35, 42, 48, 53–55], that is also published online[7]. The pattern language addresses many interesting aspects of API management, including API evolution, API representation, API quality, and RESTful conversation patterns. While most of the patterns focus on the technological aspect of API design and evolution, some patterns are also concerned with technical knowledge transfer between the API provider team and the API consumer, e.g., the *API Description* pattern [35] or the *Service Level Agreement* pattern [48]. Since the focus of our API management pattern language aims to go beyond the technical knowledge transfer and involves other stakeholders than the API consumers, we aspire to provide an addition to the microservices API pattern language.

Besides, several pattern languages exist that are partially concerned with APIs [55]. The authors of [50] discuss a middleware pattern language and present an *Interface Description* pattern. [16] presents patterns for API management, security, deployment, and adoption. However, the topic of collaboration is not addressed explicitly. Similarly, further pattern languages that address APIs with a technical focus are [11, 15, 20, 22, 30, 41, 45].

Finally, several API management books exist, that contain API management best practices, but not or only partially in form of patterns [16, 32, 37, 47].

8 DISCUSSION

This section reports six overarching findings on API management that we observed during the pattern language creation. We derived the findings from similarities or particularities of the cases in our case base. Therefore, they are specific to public, partner, and group API initiatives of primarily established organizations or SMEs from Europe. These findings aim to provide insight into current API practices and potentially provide a starting point for future studies.

(1) Most initial collaboration between the API provider and the API consumer happens through software artifacts controlled by the API management team. API consumers use

[6] https://developer.mercedes-benz.com/ (accessed 08/10/2021)

[7] https://microservice-api-patterns.org/ (accessed 08/10/2021)

the developer portal's features to discover, initially inform themselves, and contact the API provider team. Additionally, self-service options foster easy first interactions and testing of APIs. Thus, the successful collaboration between API provider and API consumers heavily depends on resources controlled by the API management team.

(2) API consumers want personal contact with the API provider before and during integrating an API. Even though API consumers often discover an API via its developer portal, in most cases, the API consumer negotiates contracts with the API provider before actually integrating the API. These contracts contain agreed-upon quality levels in the form of service-level agreements (SLAs). Thus, the API consumer can hold the API provider accountable to provide functionality that meets specific non-functional requirements, e.g., availability or performance levels. Also, the API provider can include terms of use, e.g., number of allowed calls within a time period. The use of APIs initiated solely via self-service and without a separate contract between API consumers and API provider is rare. Furthermore, after signing the contract, the API provider often supports the API consumer with integration activities.

(3) The API provider is mostly concerned with addressing the API consumers' needs. As visible in Fig. 4, we observed and documented more patterns on the right side of the API lifecycle. Significantly, the activity *"Discovery & On-boarding"* is associated with the highest number of patterns. Thus, API management searches and employs many approaches to cater to API consumer information needs and concerns.

(4) The collaboration between the API provider team and all other stakeholders is challenging. Collaboration between the API provider and the backend provider and internal stakeholders mostly focuses on quality, defect, and incident management across team, business unit, or company boundaries. The interviewees stressed the challenges of collaboration between these stakeholders, especially if APIs are not the main distribution channel of a product. In these cases, the API management team often feels the pressure of fixing issues and defects since they are the first point of consumer contact. The backend also has other tasks and only feels indirect pressure through the API provider team. Thus, the backend provider might prioritize API-related issues differently. Also, only a few approaches to standardize the collaboration between these stakeholders exist, and most collaboration relies on ad-hoc communication channels such as email. Hence, the API provider organization should adapt processes and organizational structures when starting an API initiative.

(5) The API provider has to treat the API as a product with a lifecyle. An API makes resources like functionality, data, or software products with lifecycles accessible to API consumers. However, the API itself changes due to consumer wishes, technology developments, etc., and the API changes can impact the consumer business. Therefore, the API provider needs to actively manage the API lifecycle in coordination with the backend provider and the API consumer.

(6) Strategic relevance and the structure of the consumers' organization are potential influence factors for the suitability of patterns. The API initiative cases used as a basis for eliciting the API management patterns have different characteristics that can influence the applicability of an API pattern. For example, a candidate for those factors is the question if an API provider offers a commodity functionality or data for a broad audience or if the usage of the API has a strategic impact on the consumers business and is thus probably relevant to a smaller group of consumers, i.e., within one branch. Furthermore, the organizational structure and the technical capabilities of the target consumers organization are relevant. Future research should aim at making such potential influence factors of API initiatives explicit.

9 CONCLUSION AND OUTLOOK

APIs are strategic resources at the interface between functionality providers and consumers. However, the literature on the management of APIs focuses on young, digital organizations and is too abstract to provide guidelines for API management or focuses on relatively low-level technical aspects of API design and maintenance. API management as a function embedded in an organizational context makes collaboration with stakeholders inside and outside of the organization necessary. Thus, this research endeavor aims to identify proven API management practices focusing on collaboration between stakeholders of public, partner, and group API initiatives.

To reach this goal, we apply a design science approach [28, 29] combined with grounded theory methodology [51]. We conducted 16 semi-structured interviews with 15 experts employed in the API management domain. The data analysis yields the identification of eight stakeholders of API management. Furthermore, 35 pattern candidates and 21 patterns for API management are documented. Two representative patterns of the pattern language are included in this paper. Also, six general findings regarding the collaboration of the API provider team and the pattern language are presented.

Future work will comprise several activities addressing limitations of the current version of the pattern language with the aim to evolve and validate the pattern language. First of all, the pattern language is derived from 12 cases. Thus, more interviews could lead to the detection of new pattern candidates and patterns as well as enable the validation of existing patterns. Also, the interviews currently reflect API management practices observed in mostly European organizations. Therefore, international interview partners could diversify the results.

Furthermore, the 16 interviews were conducted over a time period of half a year and, hence, provide a screenshot of current API management practices. More interviews with the same interviewees in the future would enable the collection of longitudinal data. Such interviews would allow the observation of evolution in API management practices [10].

Finally, the patterns are observed in existing initiatives. Another possibility for validating the pattern language is therefore to guide or monitor the implementation of these patterns in organizations. For instance, API providers could be guided through the application of the pattern language based on pattern workshops [10]. Such use cases would generate valuable insights for the evolution of the pattern language.

ACKNOWLEDGMENTS

We want to thank Prof. Dr. Stefan Sobernig from the Vienna University of Economics and Business (WU Vienna) for his valuable

feedback during the shepherding phase of the EuroPLoP conference. Similarly, we want to thank the participants of the EuroPLoP workshop for the feedback and discussions. Finally, we want to thank all interviewees for contributing to our research.

REFERENCES

[1] Christopher Alexander. 1972. *Notes on the Synthesis of Form.* (1 ed.). Harvard University Press.
[2] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. 1977. *A Pattern Language: Towns, Buildings, Construction.* (1 ed.). Oxford University Press.
[3] axway. [n.d.]. Axway Open Documentation - API lifecycle. https://docs.axway.com/bundle/axway-open-docs/page/docs/api_mgmt_overview/api_mgmt_lifecycle/index.html. online, accessed: 23.04.2021.
[4] Rahul C Basole. 2019. On the Evolution of Service Ecosystems: A Study of the Emerging API Economy. In *Handbook of Service Science, Volume II*. Springer, 479–495.
[5] David Bermbach and Erik Wittern. 2016. Benchmarking Web API Quality. In *Web Engineering*, Alessandro Bozzon, Philippe Cudre-Maroux, and Cesare Pautasso (Eds.). Springer International Publishing, Cham, 188–206.
[6] Gloria Bondel, Sascha Nägele, Fridolin Koch, and Florian Matthes. 2020. Barriers for the Advancement of an API Economy in the German Automotive Industry and Potential Measures to Overcome these Barriers.. In *ICEIS (1)*. 727–734.
[7] Gloria Bondel, Sven-Volker Rehm, and Florian Matthes. 2020. JUST LINKING THE DOTS? BARRIERS AND DRIVERS IN CREATING VALUE FROM APPLICATION PROGRAMMING INTERFACES. *Twenty-Eighth European Conference on Information Systems (ECIS2020) – A Virtual AIS Conference* (2020). https://aisel.aisnet.org/ecis2020_rip/70
[8] Uwe M Borghoff and Johann H Schlichter. 2000. Fundamental Principles of Distributed Systems. In *Computer-Supported Cooperative Work*. Springer, 3–85.
[9] Sabine Buckl, Alexander M. Ernst, Josef Lankes, and Florian Matthes. 2008. *Enterprise Architecture Management Pattern Catalog Version 1.0.* Technical Report 1. Software Engineering for Business Information Systems (sebis), Chair for Informatics 19, Technische Universität München, Boltzmannstraße 3, 85748 Garching b. Münnchen, Germany. Technical Report TB 0801.
[10] Sabine Buckl, Florian Matthes, Alexander W. Schneider, and Christian M. Schweda. 2013. Pattern-Based Design Research – An Iterative Research Method Balancing Rigor and Relevance. In *Design Science at the Intersection of Physical and Virtual Design*, Jan vom Brocke, Riitta Hekkala, Sudha Ram, and Matti Rossi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 73–87.
[11] Frank Buschmann, Kevlin Henney, and Douglas C Schmidt. 2007. *Pattern-Oriented Software Architecture, A Pattern Language for Distributed Computing.* Vol. 4. John Wiley & Sons.
[12] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. 1996. *Pattern-Oriented Software Architecture: A System of Patterns* (1 ed.). Wiley.
[13] Alan Cooper et al. 2004. *The inmates are running the asylum: Why high-tech products drive us crazy and how to restore the sanity.* Vol. 2. Sams Indianapolis.
[14] James O. Coplien. 1996. Software patterns. (1996).
[15] Robert Daigneau. 2012. *Service Design Patterns: fundamental design solutions for SOAP/WSDL and restful Web Services.* Addison-Wesley.
[16] Brajesh De. 2017. *API Management: An Architect's Guide to Developing and Managing APIs for Your Organization.* Apress, Berkeley, CA.
[17] Mark de Reuver, Carsten Sørensen, and Rahul C Basole. 2018. The digital platform: a research agenda. *Journal of Information Technology* 33, 2 (2018), 124–135.
[18] Doerrfeld, Bill and Pedro, Bruno and Sandoval, Kristopher and Krohn, Andreas. [n.d.]. The API Lifecycle - An Agile Process For Managing the Life of an API. https://19yw4b240vb03ws8qm25h366-wpengine.netdna-ssl.com/wp-content/uploads/theapilifecycle.pdf. ebook, accessed: 23.04.2021.
[19] Ben Eaton, Silvia Elaluf-Calderwood, Carsten Sørensen, and Youngjin Yoo. 2015. Distributed tuning of boundary resources. *MIS quarterly* 39, 1 (2015), 217–244.
[20] Thomas Erl. 2008. *SOA Design Patterns (paperback).* Pearson Education.
[21] Peter C. Evans and Rahul C. Basole. 2016. Revealing the API Ecosystem and Enterprise Strategy via Visual Analytics. *Commun. ACM* 59, 2 (Jan. 2016), 26–28. https://doi.org/10.1145/2856447
[22] Martin Fowler. 2012. *Patterns of Enterprise Application Architecture.* Addison-Wesley.
[23] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software* (1 ed.). Addison-Wesley Professional.
[24] Ahmad Ghazawneh and Ola Henfridsson. 2010. Governing third-party development through platform boundary resources. In *the International Conference on Information Systems (ICIS)*. AIS Electronic Library (AISeL), 1–18.
[25] Ahmad Ghazawneh and Ola Henfridsson. 2013. Balancing platform control and external contribution in third-party development: the boundary resources model. *Information Systems Journal* 23, 2 (2013), 173–192. https://doi.org/10.1111/j.1365-2575.2012.00406.x arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1365-2575.2012.00406.x
[26] Google Apigee. 2018. Apigee API Management Lifecycle. https://nl.devoteam.com/en/blog-post/apigee-api-management-lifecycle/. online, accessed: 23.04.2021.
[27] Ola Henfridsson, Lars Mathiassen, and Fredrik Svahn. 2014. Managing technological change in the digital age: the role of architectural frames. *Journal of Information Technology* 29, 1 (2014), 27–43.
[28] Alan R Hevner. 2007. A three cycle view of design science research. *Scandinavian journal of information systems* 19, 2 (2007), 4.
[29] Alan R Hevner, Salvatore T March, Jinsoo Park, and Sudha Ram. 2004. Design science in information systems research. *MIS quarterly* (2004), 75–105.
[30] Gregor Hohpe and Bobby Woolf. 2004. *Enterprise integration patterns: Designing, building, and deploying messaging solutions.* Addison-Wesley Professional.
[31] Anna Sigridur Islind, Tomas Lindroth, Ulrika Lundh Snis, and Carsten Sørensen. 2016. Co-creation and fine-tuning of boundary resources in small-scale platformization. In *Scandinavian conference on information systems*. Springer, 149–162.
[32] Daniel Jacobson, Greg Brail, and Dan Woods. 2012. *APIs: A Strategy Guide.* O'Reilly, The address.
[33] Kimmo Karhu, Robin Gustafsson, and Kalle Lyytinen. 2018. Exploiting and defending open digital platforms with boundary resources: Android's five platform forks. *Information Systems Research* 29, 2 (2018), 479–497.
[34] Pouya Aleatrati Khosroshahi, Matheus Hauder, Alexander W. Schneider, and Florian Matthes. 2015. *Enterprise Architecture Management Pattern Catalog Version 2.0.* Technical Report 1. Software Engineering for Business Information Systems (sebis), Chair for Informatics 19, Technische Universität München, Boltzmannstraße 3, 85748 Garching b. Münnchen, Germany. Technical Report.
[35] Daniel Lübke, Olaf Zimmermann, Cesare Pautasso, Uwe Zdun, and Mirko Stocker. 2019. Interface Evolution Patterns: Balancing Compatibility and Extensibility across Service Life Cycles. In *Proceedings of the 24th European Conference on Pattern Languages of Programs* (Irsee, Germany) *(EuroPLop '19)*. Association for Computing Machinery, New York, NY, USA, Article 15, 24 pages. https://doi.org/10.1145/3361149.3361164
[36] Nicolas Masse. [n.d.]. Full API lifecycle management: A primer. https://developers.redhat.com/blog/2019/02/25/full-api-lifecycle-management-a-primer/. online, accessed: 23.04.2021.
[37] Mehdi Medjaoui, Erik Wilde, Ronnie Mitra, and Mike Amundsen. 2018. *Continuous API Management - Making the right decisions in an evolving landscape.* Vol. 1. O'Reilly.
[38] Gerard Meszaros and Jim Doble. 1997. A pattern language for pattern writing. In *Proceedings of International Conference on Pattern languages of program design (1997)*, Vol. 131. 164.
[39] MuleSoft. [n.d.]. MuleSoft API Lifecycle. https://mulesy.com/mulesoft-api-lifecycle/. online, accessed: 23.04.2021.
[40] Alexander Osterwalder, Yves Pigneur, Gregory Bernarda, and Alan Smith. 2014. *Value proposition design: How to create products and services customers want.* John Wiley & Sons.
[41] Cesare Pautasso, Ana Ivanchikj, and Silvia Schreier. 2016. A Pattern Language for RESTful Conversations. In *Proceedings of the 21st European Conference on Pattern Languages of Programs* (Kaufbeuren, Germany) *(EuroPlop '16)*. Association for Computing Machinery, New York, NY, USA, Article 4, 22 pages. https://doi.org/10.1145/3011784.3011788
[42] C. Pautasso, O. Zimmermann, M. Amundsen, J. Lewis, and N. Josuttis. 2017. Microservices in Practice, Part 1: Reality Check and Service Design. *IEEE Software* 34, 1 (2017), 91–98. https://doi.org/10.1109/MS.2017.24
[43] John Pruitt and Tamara Adlin. 2005. *The Persona Lifecycle: Keeping People in Mind Throughout Product Design.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
[44] Redhat. [n.d.]. https://www.redhat.com/en/topics/api/what-does-an-api-gateway-do. online. Accessed: 12.02.2021.
[45] Arnon Rotem-Gal-Oz, Eric Bruno, and Udi Dahan. 2012. *SOA patterns.* Manning Shelter Island.
[46] M. Santoro, L. Vaccari, D. Mavridis, R. S. Smith, M. Posada, and D. Gattwinkel. 2019. *Web Application Programming Interfaces (APIs): General purpose standards, terms and European Commission initiatives.* Technical Report. European Commission, Louxembourg.
[47] Kai Spichale. 2017. *API-Design - Praxishandbuch für Java- und Webservice-Entwickler.* Vol. 1. dpunkt.verlag.
[48] Mirko Stocker, Olaf Zimmermann, Uwe Zdun, Daniel Lübke, and Cesare Pautasso. 2018. Interface Quality Patterns: Communicating and Improving the Quality of Microservices APIs. In *Proceedings of the 23rd European Conference on Pattern Languages of Programs* (Irsee, Germany) *(EuroPLoP '18)*. Association for Computing Machinery, New York, NY, USA, Article 10, 16 pages. https://doi.org/10.1145/3282308.3282319
[49] Ömer Uludağ, Nina-Mareike Harders, and Florian Matthes. 2019. Documenting Recurring Concerns and Patterns in Large-Scale Agile Development. In *Proceedings of the 24th European Conference on Pattern Languages of Programs* (Irsee,

Germany) *(EuroPLop '19)*. Association for Computing Machinery, New York, NY, USA, Article 27, 17 pages. https://doi.org/10.1145/3361149.3361176

[50] Markus Völter, Michael Kircher, and Uwe Zdun. 2013. *Remoting patterns: foundations of enterprise, internet and realtime distributed object middleware.* John Wiley & Sons.

[51] Manuel Wiesche, Marlen C. Jurisch, Philip W. Yetton, and Helmut Krcmar. 2017. Grounded Theory Methodology in Information Systems Research. *MIS Q.* 41, 3 (Sept. 2017), 685–701. https://doi.org/10.25300/MISQ/2017/41.3.02

[52] Youngjin Yoo, Ola Henfridsson, and Kalle Lyytinen. 2010. Research commentary—the new organizing logic of digital innovation: an agenda for information systems research. *Information systems research* 21, 4 (2010), 724–735.

[53] Olaf Zimmermann. 2017. Microservices tenets. *Computer Science-Research and Development* 32, 3 (2017), 301–310.

[54] Olaf Zimmermann, Mirko Stocker, Daniel Lübke, Cesare Pautasso, and Uwe Zdun. 2020. Introduction to microservice API patterns (MAP). *First and Second International Conference on Microservices (Microservices 2017/2019)* (2020).

[55] Olaf Zimmermann, Mirko Stocker, Daniel Lübke, and Uwe Zdun. 2017. Interface Representation Patterns: Crafting and Consuming Message-Based Remote APIs. In *Proceedings of the 22nd European Conference on Pattern Languages of Programs* (Irsee, Germany) *(EuroPLoP '17)*. Association for Computing Machinery, New York, NY, USA, Article 27, 36 pages. https://doi.org/10.1145/3147704.3147734

A INTERVIEWS

Overall, 16 interviews with 15 interview partners informed the research approach. An overview of the interviews is presented in Tab. 2 including information on the interviewees' role, the classification of the interviewees' employing organization, the organizations' size, and the duration of the interviews.

B CASE BASE

The case base is derived from the interviews at the API portal level. An overview of the case base is provided in Tab. 3.

C RELATIONS BETWEEN PATTERNS

Fig. 8 visualizes the relations between the API management patterns according to the *"Related Patterns within the Pattern Language"* section of each pattern. Also, it provides logical entry points for the API providers and consumers exploring the pattern language.

Table 2: Overview of the conducted interviews.

# Number	Industry	Role	# Employees	Duration (hh:mm:ss)	Participants
1	Financial services	Backend Developer	11 - 50	00:22:52	IV1
2	Industrial manufacturing	Internal Consulting	>100.000	00:44:09	IV2
3	Automotive	Product Owner, Product Owner	>100.000	00:48:49	IV3, IV4
4	Financial services	Software Architect	1001 - 5000	00:42:25	IV5
5	Mobility	Portfolio Manager	1001 - 5000	00:51:12	IV6
6	Insurance	Software Architect	51 - 250	00:59:28	IV7
7	Industrial manufacturing	Product Owner	>100.000	00:46:34	IV8
8	Industrial manufacturing	Software Architect	>100.000	00:47:03	IV9
9	Financial services	Software Developer	10.001 - 50.000	00:35:25	IV10
10	Financial Services	Internal Consulting	5001 - 10.000	00:50:49	IV11
11	Insurance	Integration Architect	51 - 250	00:56:29	IV12
12	Automotive	Product Owner, Product Owner	>100.000	00:51:48	IV3, IV4
13	Financial Services	Technical Lead, Product Owner	5001 - 10.000	00:55:25	IV13, IV14
14	Financial Services	Software Architect	1001 - 5000	00:50:49	IV5
15	Mobility	Portfolio Manager	1001 - 5000	00:31:58	IV6
16	Mobility	Internal Consulting	1001 - 5000	00:45:44	IV15

Table 3: Overview of the case base derived from expert interviews.

Case Number	# Interview	Architectural Openness	Maturity	Number of API Consumers
C1 (excluded)	1	Private	Development	<20
C2	2	Partner	Pilot	<20
C3	3, 12	Public & Partner	Production	>20
C4	4, 14	Public	Production	>10000
C5	4, 14	Partner	Production	>20
C6	5, 15, 16	Group	Production	na
C7	6	Group	Development	<20
C8 (excluded)	7	Private	Development	>20
C9	8	Public & Partner	Production	na
C10	9	Partner	Production	na
C11	9	Public & Partner	Production	>10000
C12	10	Partner	Pilot	<20
C13	11	Public & Partner	Production	<20
C14	13	Public & Partner	Production	>10000
C15 (excluded)	13	Private	Development	<20

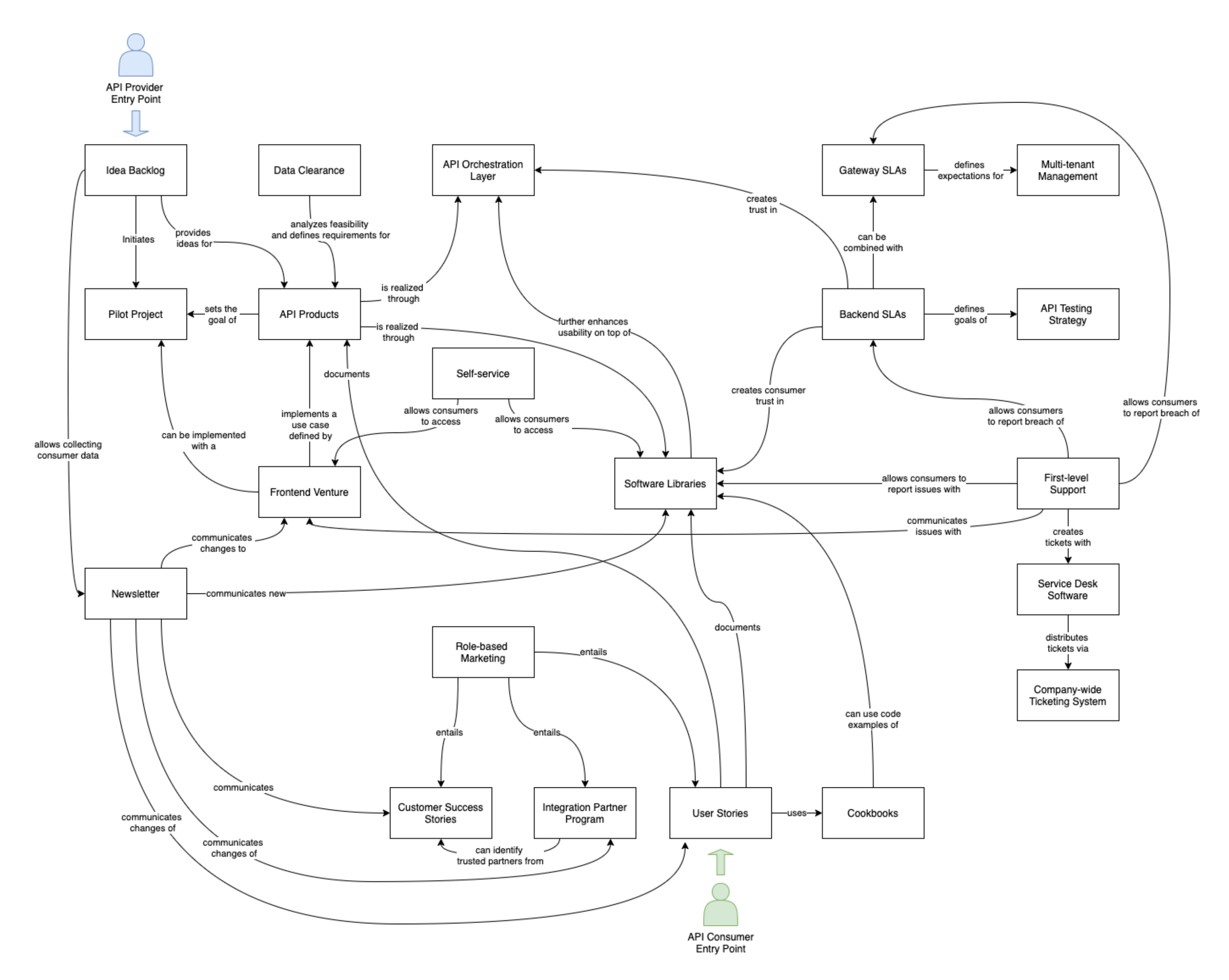


Figure 8: Relations between the API Management Patterns and entry point for API providers and consumers exploring the API management pattern language.